

Fundamental Study  
Formal validation of data-parallel programs:  
a two-component assertional proof system  
for a simple language<sup>1</sup>

Luc Bougé<sup>a,\*</sup>, David Cachera<sup>a</sup>, Yann Le Guyadec<sup>c</sup>, Gil Utard<sup>a</sup>,  
Bernard Viot<sup>b</sup>

<sup>a</sup> *LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France*

<sup>b</sup> *LIFO, Univ. Orléans, 4 Rue Léonard de Vinci, BP 6759, F-45067 Orléans Cedex 2, France*

<sup>c</sup> *VALORIA, 8 rue Montaigne, BP 1104, F-56014 Vannes, France*

Received March 1996; revised December 1996

Communicated by M. Nivat

---

**Abstract**

We present a proof system for a simple data-parallel kernel language called  $\mathcal{L}$ . This proof system is based on a two-component assertion language. We define a weakest preconditions calculus and analyze its definability properties. This calculus is used to prove the completeness of the proof system. We also present a two-phase proof methodology, yielding proofs similar to those for scalar languages. We finally discuss other approaches.

---

**Contents**

1. Introduction .....	72
2. A simple data-parallel language and its semantics .....	73
2.1. The $\mathcal{L}$ language .....	73
2.2. A natural semantics for $\mathcal{L}$ .....	75
3. A two-component assertional proof system for $\mathcal{L}$ programs .....	77
3.1. Why do we need two components? .....	77
3.2. Vector predicates and assertions .....	78
3.3. Proof system .....	80
3.4. Example .....	82
4. Weakest preconditions of $\mathcal{L}$ programs .....	83
4.1. The Definability Problem .....	84
4.2. Discussion .....	86

---

\* Corresponding author. E-mail: Luc.Bouge@lip.ens-lyon.fr.

<sup>1</sup> This work has been partly supported by the French CNRS Coordinated Research Programs on Parallelism C<sup>3</sup> and PRS, the French PRC/MRE Research Contract ParaDigne, Department of Defense DRET contract 91/1180 and the ReMaP INRIA Project.

4.3. Definability of the weakest strict preconditions of linear programs .....	87
4.4. Definability of weakest liberal preconditions .....	92
4.5. Discussion: extending the assertion language .....	94
5. Completeness of the proof system .....	95
5.1. Extending the proof of completeness to non-regular, linear programs .....	97
5.2. Extending the proof of completeness of non-plain specifications .....	99
6. A two-phase proof methodology of $\mathcal{L}$ programs .....	100
6.1. First step: syntactic labeling .....	101
6.2. Second step: proof outline .....	102
6.3. A simple example .....	103
7. Conclusion and related work .....	105
Acknowledgements .....	106
References .....	106

## 1. Introduction

Data-parallel languages have recently emerged as a major tool for large scale parallel programming. An impressive effort is currently being put on developing efficient compilers for High Performance Fortran (HPF). DPCE [9], a data-parallel extension of C primarily influenced by Thinking Machine's C\*, is currently under standardization. Our goal is to provide all these new developments with the necessary semantic bases. These bases are crucial to design safe and optimized compilers, and programming environments including parallelizing, data-distributing and debugging tools. They are also the way to safer programming techniques, so as to avoid the common waste of time and money spent in debugging.

Existing data-parallel languages, such as HPF, C\*, HyperC or MPL, include a similar core of data-parallel control structures. In previous papers, we have shown that it is possible to define a simple but representative data-parallel kernel language (the  $\mathcal{L}$  language) and to give it a formal operational [4] and denotational semantics [3].

In this paper, we define a proof system for this language, in the style of the usual Hoare's logic approach [13]. The originality of our approach lies in the treatment of the *extent of parallelism*, i.e., the subset of currently active indices at which a vector instruction is to be applied. Previous approaches led to manipulate lists of indices explicitly (either by manipulating sets of active processors [19,20] or by specifying an access sequence for each parallel variable [8]), or to consider context expressions as assertions modifiers [10]. In contrast, our proof system for  $\mathcal{L}$  describes the activity context by a vector boolean expression distinct from the usual predicates on program variables. The use of such two-component assertions is particularly well-suited to an intuitive understanding of the assertions and provides a basis for a two-phase "proof by annotations" method.

In Section 2, we give a description of the  $\mathcal{L}$  language, together with its natural semantics. Section 3 describes a sound proof system based on our two-component assertions. Section 4 deals with the definition of a weakest preconditions calculus and with the associated definability question. This weakest preconditions calculus is the key to establish the completeness of our proof system, which is treated in Section 5.

A two-phase proof methodology, yielding readable and structured proofs, is described in Section 6. As a conclusion, we present a discussion and some perspectives.

## 2. A simple data-parallel language and its semantics

### 2.1. The $\mathcal{L}$ language

In the data-parallel programming model, the basic objects are arrays with parallel access, also called *vectors*. Two kinds of actions can be applied to these objects: *componentwise* operations, or global *rearrangements*. A program is a sequential composition of such actions. Each action is associated with the set of array indices at which it is applied. An index at which an action is applied is said to be *active*. Other indices are said to be *idle*. The set of active indices is called the *activity context* or the *extent of parallelism* following the term coined for the Actus language [18]. It can be seen as a boolean array where *true* denotes activity and *false* idleness.

Observe that all usual data-parallel languages such as Actus, C\*, MPL or HPF are deterministic. Though they specify parallel accesses to data whose scheduling may be non-deterministic, the resulting semantics *is* deterministic. We are only of two exceptions: a rather special use of the send operator in C\* where the receiver may require a non-deterministic combining operator; some technical issues connected with parameter passing in idle contexts. However, we are not concerned with this level of detail in this paper, and we consider only deterministic data-parallel constructs.

The  $\mathcal{L}$  language is designed as a common kernel of data-parallel languages like C\* [21], HyperC [17] or MPL [15]. We do not consider the scalar part of these languages, mainly imported from the C language. Then, as far as this paper is concerned, we can assume that the scalar values are replicated at all locations, so that we can identify a scalar variable with a vector having the same value replicated at all components.

Also, for the sake of simplicity, we only consider integer and boolean values in this paper, and we consider that all parallel arrays share a unique geometry. This is reminiscent of the MPL language, where the common geometry of the plural variables is the physical geometry of the underlying architecture. Multiple geometries, as allowed by the shapes of C\* or the collections of HyperC, could easily be handled at the price of extra notation and case analysis. This unique geometry is captured here through a *finite* domain of indices  $\mathcal{D}$  equipped with a set of geometric operators such as shift, rotate, etc. For instance, in MPL,  $\mathcal{D}$  would be a square  $[0..1023] \times [0..1023]$ , and the associated geometric operators would be toroidal translations along the axes. The precise structure of the geometric domain  $\mathcal{D}$  and the detailed definition of the geometric operators are of little relevance here. In the examples of this paper, a one-dimensional or two-dimensional domain will be assumed. The only important point is that we assume the existence of two conversion functions:

- $x = \text{itos}(u)$  maps an index  $u$  into a scalar value  $x$ ;
- $u = \text{stoi}(x)$  maps a scalar value into an index value.

These functions are reminiscent of the `pcoord` functions in  $C^*$ , or the `iproc` function in MPL. The only hypothesis is that  $(\text{stoi} \circ \text{itos})$  is the identity function on indices:  $u = \text{stoi}(\text{itos}(u))$ . If  $\mathcal{D}$  is  $[0..N - 1]$ , then think of `itos` as embedding  $[0..N - 1]$  into the integers and `stoi` as the  $\text{mod}(N)$  operator.

All the variables of  $\mathcal{L}$  are parallel, and all the objects are vectors of scalars, with one component at each index. As a convention, the parallel objects are denoted with uppercase initial letters:  $X$ ,  $Y$ , etc. Indices are denoted  $u$ ,  $v$ , etc. The component of a parallel object  $X$  located at index  $u$  is denoted  $X|_u$ .

A vector expression  $E$  can be of the following forms:

- A vector variable  $X$ .
- A vector constant of integer or boolean type. Constant 1 denotes the vector whose all components have value 1, *True* and *False* denote the vectors whose all components are respectively true and false. Constant expression *This* denotes the vector whose value at index  $u$  is `itos(u)`: this is the `iproc` of MPL, the `.` operator of  $C^*$ .
- A componentwise combination of vector expressions: for instance,  $X + Y$ . All usual scalar operators are overloaded with their respective vector extension.
- It is useful to define an additional type of vector expressions: *conditional vector expressions*.  $(C?E:F)$  denotes the vector whose component at index  $u$  is  $E|_u$  if boolean vector expression  $C$  is true at index  $u$ , and  $F|_u$  otherwise.
- A *fetch* expression:  $E|_A$ . Consider a fixed index  $u$ . First, the vector expression  $A$  is evaluated, then the vector expression  $E$ . Finally, the result is rearranged so that the value at index  $u$  is fetched at the index which is the value of  $A$  at  $u$  (converted through function `stoi`):  $(E|_A)|_u = E|_{\text{stoi}(A|_u)}$ . In particular,  $E|_{\text{This}}$  is merely  $E$ . In MPL, this is denoted `router[A].E`. In  $C^*$  and HyperC, this is denoted  $[A]E$ .

As an example, consider a typical fragment of a one-dimensional convolution code:

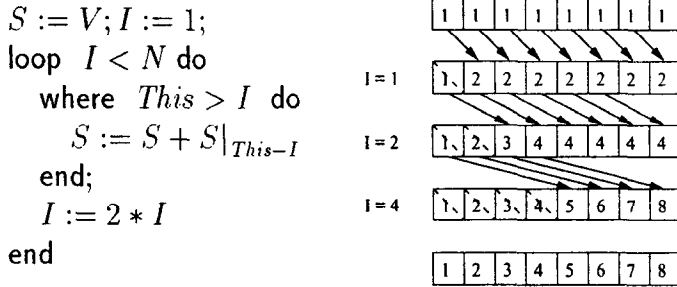
$$(2 * X + X|_{\text{This}+1} + X|_{\text{This}-1})/4.$$

Note that, strictly speaking, the  $+$  of  $\text{This}+1$  is *not* the same as the outer  $+$ , as it acts on the index domain  $\mathcal{D}$ . But there is no real reason to stress this difference any longer, and we will identify both operators in most cases. Note also that all constructs in  $\mathcal{L}$ , including communications (*fetch*), are *deterministic*. We can now list the instructions of  $\mathcal{L}$ .

**Assignment:**  $X := E$ . At each active index  $u$ , component  $X|_u$  is updated with the local value of vector expression  $E$ . Observe that  $E$  may be a fetch expression, in which case we obtain a *get* communication: *get*  $E$  from  $A$  into  $X$  is the same as  $X := E|_A$ . Observe also that we cannot express *send* communications in this simple model.

**Sequencing:**  $S; T$ . On the termination of the last action of  $S$ , the execution of the actions of  $T$  starts.

**Iteration:** *loop*  $B$  *do*  $S$  *end*. The actions of  $S$  are repeatedly executed with the current extent of parallelism, until boolean vector expression  $B$  evaluates to false at each currently active index. Observe that the activity context is not modified on executing the body, in contrast with the parallel *while* of MPL and the *whilesomewhere* of  $C^*$ .

Fig. 1. The scan  $\mathcal{L}$ -program and an execution trace.

These constructs can be expressed in  $\mathcal{L}$  by a **where** nested in a loop. Our form is therefore more general [4].

**Conditioning:** **where**  $B$  **do**  $S$  **end**. The active indices whose local value of the boolean vector expression  $B$  evaluates to false become idle during the execution of  $S$ . The other ones remain active. The initial activity context is restored on the termination of  $S$ .

The  $\mathcal{L}$  language is quite simple, but it is sufficient to express usual data-parallel algorithms. Consider for instance a *scan*, that is, a prefixed sum, using a classical logarithmic method [12]. The domain is  $\mathcal{D}[1..N]$ . Initially, the component at index  $u$ ,  $1 \leq u \leq N$ , holds an initial value  $V|_u$ , and we compute  $S$  such that  $\forall u : S|_u = \sum_{k=1}^{k=u} V|_k$ . The program in MPL-like syntax is displayed below. The  $\text{XnetW}[i]$  construct expresses a fetch of indices at distance  $i$  towards low indices.

```

S=V; i=1;
while (i < N) do {                               /*scalar while*/
  if (iproc > i)                                   /*plural if*/
    S += XnetW[i].S;
  i *= 2;
}

```

Its translation in  $\mathcal{L}$  is displayed in Fig. 1. As  $\mathcal{L}$  has no scalar variables, we translate the MPL scalar variable  $i$  into a vector variable  $I$  whose all components hold the common value  $i$ . The execution trace shows the successive values of the vector  $S$  during the computation, surrounded by its input and output value. Crossed values tag components inactive in the inner **where** construct. Arrows denote fetched values.

## 2.2. A natural semantics for $\mathcal{L}$

We describe the semantics of  $\mathcal{L}$  in the style of the natural semantics by induction on the syntax of  $\mathcal{L}$  programs.

An *environment*  $\sigma$  is a function from identifiers to vector values. The set of environments is denoted by  $Env$ . For convenience, we extend the environment functions to the

parallel expressions:  $\sigma(E)$  denotes the value obtained by evaluating parallel expression  $E$  in environment  $\sigma$ . Here are the most interesting rules:

- $\sigma(\text{This})|_u = \text{itos}(u)$
- $\sigma(E + F)|_u = \sigma(E)|_u + \sigma(F)|_u$
- $\sigma(C?E:F)|_u = \begin{cases} \sigma(E)|_u & \text{if } \sigma(C)|_u \text{ is true} \\ \sigma(F)|_u & \text{otherwise} \end{cases}$
- $\sigma(E|_A)|_u = \sigma(E)|_{\text{stoi}(\sigma(A)|_u)}$

Let  $\sigma$  be an environment,  $X$  a vector variable and  $V$  a vector value. We denote by  $\sigma[X \leftarrow V]$  the new environment  $\sigma'$  where  $\sigma'(X) = V$  and  $\sigma'(Y) = \sigma(Y)$  for all  $Y \neq X$ .

A *context*  $c$  is a boolean vector. It specifies the activity at each index. We distinguish a particular context denoted by *True* where all components have the boolean value true. For convenience, we define the activity predicate  $\text{active}_c$ :  $\text{active}_c(u) \equiv c|_u$ .

A *state*  $s$  is a pair  $(\sigma, c)$  made of an environment  $\sigma$  and a context  $c$ . We distinguish an additional special state,  $\perp$ , to denote non-termination.

The semantics  $\llbracket S \rrbracket$  of a program  $S$  is a *strict* function from states to states:  $\llbracket S \rrbracket(\perp) = \perp$ . We extend the function  $\llbracket S \rrbracket$  to sets of states as usual. Observe that it would not be difficult to extend this work to non-deterministic programs by defining  $\llbracket S \rrbracket$  to be a function from states to sets of states. As we are only concerned with deterministic data-parallel languages, we disregard this extension in this paper.

*Assignment.* At each active index, the component of the parallel variable is updated with the new value.

$$\llbracket X := E \rrbracket(\sigma, c) = (\sigma', c),$$

with  $\sigma' = \sigma[X \leftarrow V]$  where  $V|_u = \sigma(E)|_u$  if  $\text{active}_c(u)$ , and  $V|_u = \sigma(X)|_u$  otherwise. The activity context is preserved.

*Sequencing.* Sequential composition is functional composition.

$$\llbracket S; T \rrbracket(\sigma, c) = \llbracket T \rrbracket(\llbracket S \rrbracket(\sigma, c)).$$

*Iteration.* Iteration is expressed by classical loop unfolding. It terminates when the boolean expression  $B$  evaluates to false at each active index. We have the relation

$$\llbracket \text{loop } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = \begin{cases} \llbracket \text{loop } B \text{ do } S \text{ end} \rrbracket(\llbracket S \rrbracket(\sigma, c)) & \text{if } \exists u : (\text{active}_c(u) \wedge \sigma(B)|_u), \\ (\sigma, c) & \text{otherwise.} \end{cases}$$

If the unfolding does not terminates, then we take the usual convention:

$$\llbracket \text{loop } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = \perp.$$

To see that this is well-defined, we can proceed exactly as in the usual case. Define  $\phi_k(\sigma, c)$  to be the final state of  $\text{loop } B \text{ do } S \text{ end}$  after evaluating at most  $k$  times the test. We have  $\phi_0(\sigma, c) = \perp$ , and

$$\phi_{k+1}(\sigma, c) = \begin{cases} \phi_k(\llbracket S \rrbracket(\sigma, c), c) & \text{if } \exists u : (\text{active}_c(u) \wedge \sigma(B)|_u), \\ (\sigma, c) & \text{otherwise.} \end{cases}$$

It can be shown that if  $\phi_k(\sigma, c) \neq \perp$ , then  $\phi_{k+1}(\sigma, c) = \phi_k(\sigma, c)$ . Then, we can define

$$\llbracket \text{loop } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = \bigsqcup_k \phi_k(\sigma, c)$$

where  $\bigsqcup_k$  denotes the least upper bound for the flat partial order ( $\perp \leq x$ ,  $x \not\leq y$ ), which clearly satisfies the relation above.

**Conditioning.** The denotation of a **where** construct is the denotation of its body with a new context. The new context is the conjunction of the previous one with the value of the conditioning expression  $B$ .

$$\llbracket \text{where } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = (\sigma', c),$$

with  $\llbracket S \rrbracket(\sigma, c \wedge \sigma(B)) = (\sigma', c')$ . The value of  $c'$  is ignored here.

**Remark.** In this language, the activity context is preserved by terminating executions: for any program  $S$  such that  $\llbracket S \rrbracket(\sigma, c) = (\sigma', c')$ , we have  $c = c'$ . It is no longer true for the extended version of  $\mathcal{L}$  defined in [5], which includes a data-parallel **break**-like construct.

### 3. A two-component assertional proof system for $\mathcal{L}$ programs

#### 3.1. Why do we need two components?

We define a proof system for the partial correctness of  $\mathcal{L}$  programs in the lines of [1]. A specification is denoted by a formula  $\{Pre\} S \{Post\}$  where  $S$  is the program text, and  $Pre$  and  $Post$  are logical assertions on variables of  $S$ . This formula means that, if precondition  $Pre$  is satisfied in the initial state of program  $S$ , and if  $S$  terminates, then postcondition  $Post$  is satisfied in the final state. A proof system gives a formal method to derive such specification formulas by syntax-directed induction on programs. Axioms correspond to statements, and inference rules to control structures. Then, proving that a program meets its specification is equivalent to derive the specification formula  $\{Pre\} S \{Post\}$  in the proof system. A crucial property of axiomatic semantics in the usual sequential case is *compositionality*. To achieve this goal, the assertion language has to include sufficient information on the values of the variables. Similarly, our assertion language has to include some information about the current activity context as well.

Our proposition is to define two-component assertions  $\{P, C\}$ , where  $P$  is a predicate on the vector variables of the program, and  $C$  is a boolean vector expression which evaluates into the current activity context. To see the benefits of this approach, consider a typical  $\mathcal{L}$  program and its annotation as shown in Fig. 2. Assertion  $\{..., True\}$  means that all indices are active. Assertion  $\{..., B_1 \wedge B_2\}$  means that the active indices are precisely those indices  $u$  such that  $B_1|_u \wedge B_2|_u$  is true. We will show in this paper that such an annotation is always valid if no variable of the context expressions  $B_i$  are

<pre> ... where B<sub>1</sub> do   ...   where B<sub>2</sub> do     ...   end   ...   where B<sub>3</sub> do     ...   end   ... end ... </pre>	<pre> {..., True} ... where B<sub>1</sub> do   {..., B<sub>1</sub>}   ...   where B<sub>2</sub> do     {..., B<sub>1</sub> ∧ B<sub>2</sub>}     ...   end   ...   where B<sub>3</sub> do     {..., B<sub>1</sub> ∧ B<sub>3</sub>}     ...   end   {..., B<sub>1</sub>}   ... end {..., True} ... </pre>
---	---

Fig. 2. A typical  $\mathcal{L}$  program and its annotation by two-component assertions.

modified by the program. The proof of a data-parallel program with our method can thus be factorized into two phases:

- (i) partially annotate with context expressions;
- (ii) complete each annotation with its predicate component.

Part 1 is mostly straightforward up to variable conflicts. Part 2 is very similar to proving sequential programs. Proving a data-parallel program in our approach looks thus very much like proving ordinary scalar programs. In particular, the complexity of the proof does not depend on the size of the underlying domain.

### 3.2. Vector predicates and assertions

The structure of predicates on vector variables has to be made precise here.

- An *index expression* is either an index variable ( $u, v$ , etc), or an index constant (0, 1, etc, if a one-dimensional domain is assumed for instance), or a combination of index expressions with a geometric operator ( $u + 1, u - 1$  in the one-dimensional case), or the function *stoi* applied to a scalar expression.
- A *scalar expression* is either a scalar variable ( $x, y, \dots$ ), or a scalar constant (0, *true*, *false*, etc.), or a combination of scalar expressions with some scalar operator, or a vector expression of the programming language subscripted by an index expression  $E|_u$ , or the function *itos* applied to an index expression.
- A *formula* is either a scalar expression of boolean type, or the combination of formulas with logical operators, or a formula quantified on a scalar variable, or an index variable (in this last case, the quantification implicitly ranges on the index domain  $\mathcal{D}$ ).
- A (vector) predicate is a formula which is closed with respect to all index and scalar variables. External universal quantification is usually left implicit.



For instance, the following are vector predicates:

$\forall u : X _u = 0$	All components of $X$ have value 0
$\forall u : \forall v : X _u = X _v$	All components of $X$ are equal
$\forall u : X _u = Y _u$	$X$ and $Y$ have the same value at each index
$\forall u : \text{even}(u) \Rightarrow (X _{u+1} = X _u)$	At all even indices, the right component is the same as the local one
$\exists i : \forall u : X _u = i$	$X$ is a constant vector

Observe there is no quantification on vector variables in vector predicates. Observe also that  $X = Y$  is *not* a predicate, but a boolean vector expression defined pointwise. The usual equality predicate is  $\forall u : X|_u = Y|_u$ , which we denote by  $X \equiv Y$ .

Because a vector predicate is a formula closed with respect to the index and scalar variables, we can define its truth value with respect to an environment in the usual way. Observe that scalar variables range over integers or booleans, whereas index variables range over  $\mathcal{D}$ . If the predicate  $P$  is true in the environment  $\sigma$ , then we write  $\sigma \models P$ . We are now in position to define the validity of an assertion in a program state.

**Definition 1 (Satisfiability).** Let  $(\sigma, c)$  be a state,  $\{P, C\}$  an assertion. We say that the state  $(\sigma, c)$  satisfies the assertion  $\{P, C\}$ , and write  $(\sigma, c) \models \{P, C\}$ , if  $\sigma \models P$  and  $\sigma(C) = c$ . By convention,  $\perp$  satisfies any assertion. The set of states satisfying  $\{P, C\}$  is denoted by  $\llbracket \{P, C\} \rrbracket$ .

Consider two assertions  $\{P, C\}$  and  $\{Q, D\}$ . We say that  $\{P, C\} \Rightarrow \{Q, D\}$  if for a state  $(\sigma, c)$ ,  $\{Q, D\}$  holds as soon as  $\{P, C\}$  holds:

- (i) if  $\sigma \models P$ , then  $\sigma \models Q$ ;
- (ii) if  $\sigma \models P$  and  $\sigma(C) = c$ , then  $\sigma(D) = c$ .

**Definition 2 (Assertion implication).** Let  $\{P, C\}$  and  $\{Q, D\}$  be two assertions. We say that assertion  $\{P, C\}$  implies assertion  $\{Q, D\}$  w.r.t. context, written  $\{P, C\} \Rightarrow \{Q, D\}$ , if for any environment  $\sigma$ ,  $\sigma \models P \Rightarrow Q$  and  $\sigma \models P \Rightarrow \forall u : (C|_u = D|_u)$ .

**Proposition 3.** Let  $\{P, C\}$  and  $\{Q, D\}$  be two assertions. Then,  $\{P, C\} \Rightarrow \{Q, D\}$  iff  $\llbracket \{P, C\} \rrbracket \subseteq \llbracket \{Q, D\} \rrbracket$ .

We introduce a substitution mechanism for vector variables. Let  $P$  be a predicate or any vector expression,  $X$  a vector variable, and  $E$  a vector expression.  $P[E/X]$  denotes the predicate, or expression, obtained by substituting all the occurrences of  $X$  in  $P$  with  $E$ . Note that all vector variables are free by definition of our assertion language. The key result is that the usual substitution lemma [1] extends to this new setting.

**Lemma 4.** Let  $P$  be a predicate on vector variables,  $X$  a vector variable, and  $E$  a vector expression.

$$\sigma \models P[E/X] \text{ iff } \sigma[X \leftarrow \sigma(E)] \models P.$$

**Proof.** This is easily proved by induction on the structure of the predicates and the expressions. The crucial point is that we only consider here the substitution of a *vector*  $X$  as a whole, in contrast with [1] where the substitution of a particular component  $X[u]$  is supported.

### 3.3. Proof system

We can define the validity of a specification of a  $\mathcal{L}$  program with respect to its natural semantics. Because  $\perp$  satisfies any assertion, our definition of validity is relative to partial correctness, i.e. we are not concerned by the proof of the program termination.

**Definition 5 (Validity).** Let  $S$  be a  $\mathcal{L}$  program,  $\{P, C\}$  and  $\{Q, D\}$  two assertions. We say that the specification  $\{P, C\}S\{Q, D\}$  is valid, denoted by  $\models \{P, C\}S\{Q, D\}$ , if, for any state  $(\sigma, c)$ ,

$$(\sigma, c) \models \{P, C\} \Rightarrow \llbracket S \rrbracket(\sigma, c) \models \{Q, D\}.$$

Our goal is to catch valid formulas through a finite set of simple axioms and inference rules. Unfortunately, this turns out to be more difficult than in the usual case.

Consider the assignment statement  $x := e$  of usual sequential languages. The associated *backward* axiom is  $\{P[e/x]\}x := e\{P\}$ . A direct generalization to the  $\mathcal{L}$  language should be

$$\{P[(C?E:X)/X], C\}X := E\{P, C\}.$$

In this axiom, we express that the local assignment  $X|_u := E|_u$  is carried out only at the *active* indices, that is those indices where  $C$  evaluates to true. Thus, the former value of  $X|_u$  is  $E|_u$  if  $C|_u$  is true, and it is left unchanged otherwise. This is exactly  $(C?E:X)|_u$  according to our definition in Section 2.1.

Unfortunately, this generalization is not correct in all cases. The specification

$$\{\text{true}, Y = 2\}X := 1\{\text{true}, Y = 2\}$$

is valid. Yet,

$$\{\text{true}, X = 2\}X := 1\{\text{true}, X = 2\}$$

is not valid: the (unchanged) activity context is no longer described by the boolean expression  $X = 2$  after the assignment  $X := 1$ , since variable  $X$  has been modified. This generalization is correct *only* if the variables of the current context expression  $C$  are not modified by executing the assignment  $X := E$ .

Following the notation of [1], let  $\text{Var}(S)$  be the set of variables appearing in the program  $S$ . Let  $\text{Change}(S)$  be the set of variables appearing on the left hand side

of assignments in the program  $S$ . Only these variables can have their values changed by executing  $S$ . Let  $Var(C)$  be the set of variables appearing in the expression  $C$ . The value of  $C$  depends on these variables only. We describe below a restricted proof system where we always assume that context expressions are not modified by program bodies:  $Change(S) \cap Var(C) = \emptyset$ .

**Rule 1** (Assignment:  $X := E$ ). *We extend the usual backward axiom by taking into consideration that the vector variable  $X$  is modified only at the active indices.*

$$\frac{X \notin Var(C)}{\{P[(C?E:X)/X], C\} X := E \{P, C\}}$$

For instance, consider the postcondition  $\{\forall u : (Y|_u = X|_u), Y = 2\}$ : vectors  $X$  and  $Y$  have all their components equal, and the active indices are precisely those such that the component of  $Y$  has value 2. The following specification is valid:

$$\{\forall u : (Y|_u = ((Y|_u = 2)?1 : X|_u), Y = 2\}$$

$$X := 1$$

$$\{\forall u : (Y|_u = X|_u), Y = 2\}$$

It boils down to:

$$\{\forall u : (Y|_u = 2 \Rightarrow Y|_u = 1) \wedge (Y|_u \neq 2 \Rightarrow Y|_u = X|_u), Y = 2\}$$

$$X := 1$$

$$\{\forall u : (Y|_u = X|_u), Y = 2\}$$

that is

$$\{\forall u : (Y|_u \neq 2) \wedge (Y|_u = X|_u), Y = 2\}$$

$$X := 1$$

$$\{\forall u : (Y|_u = X|_u), Y = 2\}$$

**Rule 2** (Sequencing:  $S; T$ ). *It is a straightforward generalization of the usual case.*

$$\frac{\{P, C\} S \{R, C\}, \{R, C\} T \{Q, C\}}{\{P, C\} S; T \{Q, C\}}$$

**Rule 3** (Iteration:  $\text{loop } B \text{ do } S \text{ end}$ ). *The usual loop invariant assertion has here to be invariant with respect to both the variables values and the activity context.*

$$\frac{\{I \wedge \exists u : (C|_u \wedge B|_u), C\} S \{I, C\}}{\{I, C\} \text{loop } B \text{ do } S \text{ end} \{I \wedge \forall u : (C|_u \Rightarrow \neg B|_u), C\}}$$

**Rule 4** (Conditioning: *where B do S end*). *Following the natural semantics, the context part is the conjunction of the previous context expression and the boolean vector guard of the conditioning construct.*

$$\frac{\{P, (C \wedge B)\} S \{Q, D\}, \text{Change}(S) \cap \text{Var}(C) = \emptyset}{\{P, C\} \text{ where } B \text{ do } S \text{ end } \{Q, C\}}.$$

Consider for instance the following valid specification:

$$\{\forall u: X|_u \geq 0, X \geq 1\} X := X + 1 \{\forall u: X|_u \geq 0, X \geq 2\}.$$

The rule above applies and yields:

$$\{\forall u: X|_u \geq 0, \text{True}\}$$

where  $X \geq 1$  do  $X := X + 1$  end

$$\{\forall u: X|_u \geq 0, \text{True}\}$$

Observe that the resulting activity context expression  $D$  is ignored, very much like the resulting activity context  $c'$  in the natural semantics (Section 2.2) and that we do not need to assume  $\text{Change}(S) \cap \text{Var}(B) = \emptyset$ .

**Rule 5** (Consequence rule). *Following Definition 2, we can state the consequence rule:*

$$\frac{\{P, C\} \Rightarrow \{P_1, C_1\}, \{P_1, C_1\} S \{Q_1, D_1\}, \{Q_1, D_1\} \Rightarrow \{Q, D\}}{\{P, C\} S \{Q, D\}}$$

This rule allows us to strengthen preconditions, and to weaken postconditions of specifications.

If a specification  $\{P, C\} S \{Q, D\}$  can be derived in this proof system, we write

$$\vdash \{P, C\} S \{Q, D\}$$

**Proposition 6** (Soundness). *This proof system is sound: if  $\vdash \{P, C\} S \{Q, D\}$  then  $\models \{P, C\} S \{Q, D\}$ .*

### 3.4. Example

Let us prove the correctness of the small program given in Section 2.1.

```

S := V; I := 1;
loop I < N do
  where This > I do
    S := S + S|This-I
  end;
  I := 2 * I
end

```

To find an invariant assertion, we express that only those components whose index  $u$  is greater than scalar value  $i$  can fetch an additional value  $S|_{u-i}$ . The other components already hold the final result. Let us define the following vector predicate: *Const* expresses that vector  $I$  is a constant vector whose value is  $i$ ; *Inv* expresses that the partial sum is already computed from index 1 to  $i$ . The predicate part of the invariant assertion is  $Const \wedge Inv$ . For the sake of conciseness,  $V|_u^v$  denotes  $\sum_{k=u}^{k=v} V|_k$  and  $i$  denotes  $I|_1$ . We drop all *stoi/itos* conversions for simplicity.

$$Const \equiv \forall u : (I|_u = i)$$

$$Inv \equiv \forall u : (1 \leq u \leq i \Rightarrow S|_u = V|_1^u) \\ \wedge \quad (i < u \leq N \Rightarrow S|_u = V|_{u-i+1}^u)$$

$$Pred \equiv Const \wedge Inv$$

The sketch of the partial correctness proof is expressed by the program annotated with assertions shown in Fig. 3. The steps of the proof derivation are to check the *correctness* of invariant assertion  $\{Pred, True\}$ :  $(b) \Rightarrow (c)$ , and that it *implies* the final specification:  $(j) \Rightarrow (k)$ . In assertion  $(e)$ , note how context expression  $This > I$  is generated by the conditioning expression of the enclosing **where** block. Assertion  $(e)$  is obtained from assertion  $(f)$  by substituting  $S$  with  $(This > I ? (S + S|_{This-I}) : S)$ . That is,  $S_u$  is substituted with  $S|_u + S|_{u-i}$  if  $u > i$ . The new boundary  $2 * i$  (that is,  $2 * I|_1$ ) substituted in the invariant comes from the assignment statement for counter  $I$  in context *True*.

#### 4. Weakest preconditions of $\mathcal{L}$ programs

The *weakest liberal precondition* of a program  $S$  with respect to a set of states  $\mathcal{E}$ ,  $wlp(S, \mathcal{E})$ , is the set of all the states  $s$  such that, whenever  $S$  is activated in  $s$  and *properly terminates*, the resulting state is in  $\mathcal{E}$ . In contrast, the *weakest strict precondition* of  $S$  with respect to  $\mathcal{E}$ , (or *weakest precondition* for short when no confusion may arise),  $wp(S, \mathcal{E})$ , is the set of all the states  $s$  such that whenever  $S$  is activated in  $s \neq \perp$ , it is guaranteed to *terminate* and the final state is in  $\mathcal{E}$ . For the sake of conciseness, we define the *convergence predicate*  $conv(S, s)$  by

$$conv(S, s) \equiv s \neq \perp \Rightarrow \llbracket S \rrbracket(s) \neq \perp.$$

**Definition 7** (*Weakest preconditions*). Let  $\mathcal{E}$  be a set of states,  $S$  a  $\mathcal{L}$ -program. We define the *weakest liberal preconditions* as

$$wlp(S, \mathcal{E}) = \{s \mid \llbracket S \rrbracket(s) \in \mathcal{E} \cup \{\perp\}\}$$

and the *weakest (strict) preconditions* as

$$wp(S, \mathcal{E}) = wlp(S, \mathcal{E}) \cap \{s \mid conv(S, s)\}.$$

```

      (a) {true, True}
S := V; I := 1;
      (b) {Pred, True}
loop I < N do
      (c) {Pred ∧ i < N, True}
      (d) {Const ∧
          ∀u : (1 ≤ u ≤ i ⇒ S|u = V|1u)
          ∧ (i < u ≤ 2 * i ⇒ S|u + S|u-i = V|1u)
          ∧ (2 * i < u ≤ N ⇒ S|u + S|u-i = V|u-i*2+1u),
          True}
      where This > I do
      (e) {Const ∧
          ∀u : (1 ≤ u ≤ i ⇒ S|u = V|1u)
          ∧ (i < u ≤ 2 * i ⇒ S|u + S|u-i = V|1u)
          ∧ (2 * i < u ≤ N ⇒ S|u + S|u-i = V|u-i*2+1u),
          This > I}
      S := S + S|This-I

      (f) {Const ∧
          ∀u : (1 ≤ u ≤ i * 2 ⇒ S|u = V|1u)
          ∧ (i * 2 < u ≤ N ⇒ S|u = V|u-i*2+1u),
          This > I}
      end;
      (g) {Const ∧
          ∀u : (1 ≤ u ≤ i * 2 ⇒ S|u = V|1u)
          ∧ (i * 2 < u ≤ N ⇒ S|u = V|u-i*2+1u),
          True}
      I := 2 * I
      (h) {Const ∧
          ∀u : (1 ≤ u ≤ i ⇒ S|u = V|1u)
          ∧ (i < u ≤ N ⇒ S|u = V|u-i+1u),
          True}
      (i) {Pred, True}
      end
      (j) {Pred ∧ i ≥ N, True}
      (k) {∀u : 1 ≤ u ≤ N ⇒ S|u = V|1u, True}

```

Reminder: We write  $i$  for  $I|_1$  as *Const* expresses that  $I$  is a constant vector. We drop all *stoi/itos* conversions.

Fig. 3. The annotated *scan* program.

**Lemma 8** (Consequence Lemma).

$$\models \{P, C\} S \{Q, D\} \quad \text{iff} \quad \llbracket \{P, C\} \rrbracket \subseteq \text{wlp}(S, \{Q, D\}).$$

**Proof.** Assume  $\models \{P, C\} S \{Q, D\}$ . Then, for all  $(\sigma, c) \in \llbracket \{P, C\} \rrbracket$ , we have  $\llbracket S \rrbracket(\sigma, c) \models \{Q, D\}$ . Thus,  $\llbracket \{P, C\} \rrbracket \subseteq \text{wlp}(S, \{Q, D\})$ .

Conversely, let us assume  $\llbracket \{P, C\} \rrbracket \subseteq \text{wlp}(S, \{Q, D\})$ . Consider  $(\sigma, c) \in \llbracket \{P, C\} \rrbracket$ . By hypothesis,  $(\sigma, c) \in \text{wlp}(S, \{Q, D\})$ . Thus,  $\llbracket S \rrbracket(\sigma, c) \models \{Q, D\}$ .

#### 4.1. The Definability Problem

We restrict our study of the weakest preconditions to those subsets of states which can be described by some assertion. In classical Hoare's logic, the *Definability* property

states that the weakest preconditions of a program, with respect to a set of states described by some assertion, can itself be described by some assertion. In our framework, the Definability Problem can be stated as follows:

*Given a program  $S$  and an assertion  $\{Q, D\}$ , does there exist any assertion  $\{P, C\}$  such that*

$$wlp(S, \{Q, D\}) = \{P, C\} \quad (\text{resp. } wp(S, \{Q, D\}) = \{P, C\})$$

*If so, can it be expressed from  $S, Q, D$ ?*

It can be shown [1] that this property holds for the classical Hoare's logic under some assumptions on the expressivity of the assertion language. In our setting, the form of the assertions introduces a limitation on their expressive power. Specifying the context by an additional independent component lets it depend functionally on the the variable values. The price to pay for simpler proofs is a more complex theory. Alternative approaches are discussed in Section 4.5.

**Fact 9** (Restricted expressive power of assertions). *Let  $\{P, C\}$  be an assertion. For any environment  $\sigma$ , there exists at most one activity context  $c$  such that  $(\sigma, c) \in \llbracket \{P, C\} \rrbracket$ , namely  $c = \sigma(C)$ .*

An easy consequence is that the weakest liberal preconditions of some  $\mathcal{L}$  programs cannot be defined by any assertion. This is a major difference with the usual case. Consider, for instance,

$$S \equiv \text{loop } \text{True} \text{ do } X := X \text{ end}$$

Consider postcondition  $\{\text{true}, \text{True}\}$ . This postcondition is satisfied either if  $S$  terminates with context  $\text{True}$  or if  $S$  diverges. The former cannot occur because of the semantics of the loop construct. The latter occurs if and only if there is at least one active index, i.e., context  $c$  may satisfy the condition  $\exists u : c|_u = \text{true}$ . We have thus

$$wlp(S, \{\text{true}, \text{True}\}) = \{(\sigma, c) \mid \exists u : c|_u = \text{true}\} \cup \{\perp\}.$$

As two different activity contexts  $c$  may produce a *divergence* for the same environment  $\sigma$ , this set of states cannot be defined by any assertion by the fact above. In contrast, the weakest strict preconditions exclude divergence, and one can check that

$$wp(S, \{\text{true}, \text{True}\}) = \{\perp\}.$$

Now, the set of states  $\{\perp\}$  can be for instance defined as  $\llbracket \{\text{false}, \text{False}\} \rrbracket$ .

Unfortunately, the definability property does not hold for the weakest preconditions either. Let

$$S \equiv X := X + 1.$$

Consider  $wp(S, \{Q, D\})$ , with

$$Q \equiv (\forall u : X|_u = 1) \vee (\forall u : X|_u = 2) \quad \text{and} \quad D \equiv (X = 2).$$

Let  $\sigma$  be an initial state such that  $\forall u : \sigma(X)|_u = 1$ . Let  $\sigma'$  be the corresponding final state. If all indices are active, then the assignment occurs everywhere, and  $\forall u : \sigma'(X)|_u = 2$ . Thus  $D$  evaluates to *True* in  $\sigma'$  and the final state satisfies  $\{Q, D\}$ . If all indices are idle, then nothing changes:  $\forall u : \sigma'(X)|_u = 1$ . Thus  $D$  evaluates to *False* in  $\sigma'$  and the final state satisfies  $\{Q, D\}$ , too. We thus have

$$(\sigma, \text{True}) \in wp(S, \{Q, D\}) \quad \text{and} \quad (\sigma, \text{False}) \in wp(S, \{Q, D\}).$$

If  $wp(S, \{Q, D\})$  was described by some assertion  $\{P, C\}$ , then  $\sigma(C)$  should be equal both to *True* and to *False*. This is thus impossible. However, we shall see that a suitable restriction on the syntax of context expressions yields the Definability Property for weakest preconditions.

#### 4.2. Discussion

These preliminary remarks show that our choice of two-component assertions  $\{P, C\}$  leads to difficulties when the variables of  $C$  are modified by the program

$$wlp(X := X + 1, \{Q, X = 2\})$$

with  $Q \equiv (\forall u : X|_u = 1) \vee (\forall u : X|_u = 2)$  is not definable whereas

$$wlp(Y := Y + 1, \{Q, X = 2\})$$

is definable, as shown later.

Two alternatives can be considered here.

- (i) Change the assertion language to support more general dependencies between  $\sigma$  and  $c$ . In our setting, we consider explicit functional dependencies only:

$$(\sigma, c) \models \{P, C\} \quad \text{iff} \quad \sigma \models P \quad \text{and} \quad c = \sigma(C).$$

A possible extension would be to consider implicit logical dependency: introduce a new name, say  $\#$  after the Actus terminology [8], to denote the current context as a value in the environment. We then can consider generalized assertions of the form  $\mathcal{P}(\#)$ , with

$$(\sigma, c) \models \mathcal{P}(\#) \quad \text{iff} \quad \sigma[\# \leftarrow c] \models \mathcal{P}(\#).$$

This supports sets of context: take for instance

$$\mathcal{P}(\#) = \forall u : (\#|_u = \text{true}) \vee \forall u : (\#|_u = \text{false}).$$

A two-component assertion  $\{P, C\}$  is nothing more than the special case

$$\mathcal{P}(\#) \equiv P \wedge \forall u : (\#|_u = C|_u).$$



This direction has been explored by Le Guyadec and Viot in [14]. We discuss its relationship with our work in Section 4.5. The main drawback is that it does not support a two-phase proof methodology any more.

- (ii) Keep this assertion language and improve the proof system to circumvent this lack of definability. This is done in Section 5, using an additional rule to handle hidden variables in assertions. We show in Section 6 that it actually leads to a two-phase proof methodology where the context expressions and predicates on vector variables are handled separately.

#### 4.3. Definability of the weakest strict preconditions of linear programs

For now on, we restrict ourselves to the most basic case, which consists of  $\mathcal{L}$  programs without loops, and context expressions not modified by programs. The extension to the general case is discussed in the end of this section. The following notion will be useful in this section.

**Definition 10** (*Linear  $\mathcal{L}$  programs*). A  $\mathcal{L}$  program  $S$  is *linear* if it is made of assignments, sequencing and conditioning only.

Note that a linear program may not diverge, and that its weakest liberal preconditions are thus identical to its weakest strict preconditions. We can thus safely drop the distinction.

**Definition 11** (*Plain specification*). A pair  $(S, \{Q, D\})$  is said to be *plain* if we have  $\text{Var}(D) \cap \text{Change}(S) = \emptyset$ . A specification formula  $\{P, C\} S \{Q, D\}$  is said to be *plain* if  $(S, \{Q, D\})$  is plain.

We call the weakest preconditions of a plain pair  $(S, \{Q, D\})$  a *plain weakest precondition*.

##### 4.3.1. Weakest preconditions of basic constructs

Let us first consider the weakest precondition of assignment and sequential composition.

**Proposition 12** (*Assignment*). If  $X \notin \text{Var}(D)$ , then

$$\text{wp}(X := E, \{Q, D\}) = \{Q[(D?E : X)/X], D\}.$$

**Proof.** Let  $(\sigma, c) \in \text{wlp}(X := E, \{Q, D\})$ . Assume that  $\llbracket X := E \rrbracket(\sigma, c) = (\sigma', c)$ . Then,  $(\sigma', c) \in \llbracket \{Q, D\} \rrbracket$ . As  $X \notin \text{Var}(D)$ , we have  $\sigma(D) = \sigma'(D) = c$ . By definition,  $\sigma' = \sigma[X \leftarrow \sigma(D?E : X)] \models Q$ . By the Substitution Lemma, we deduce  $\sigma \models Q[(D?E : X)/X]$ .

Conversely, let  $(\sigma, c) \in \llbracket \{Q[(D?E : X)/X], D\} \rrbracket$ . By definition, we have  $\sigma \models Q[(D?E : X)/X]$  and  $\sigma(D) = c$ . Let  $(\sigma', c) = \llbracket X := E \rrbracket(\sigma, c)$ . By definition,  $\sigma' = \sigma[X \leftarrow \sigma(D?E : X)]$ . By the Substitution Lemma, as  $\sigma \models Q[(D?E : X)/X]$ , we deduce  $\sigma' \models Q$ . As above,  $\sigma'(D) = \sigma(D) = c$  and  $(\sigma', c) \in \llbracket \{Q, D\} \rrbracket$ .

**Proposition 13** (Sequential composition).

$$wp(S; T, \{Q, D\}) = wp(S, wp(T, \{Q, D\})).$$

**Proof.** By definition

$$\begin{aligned} wp(S; T, \{Q, D\}) &= \{s \mid \llbracket S; T \rrbracket(s) \in \llbracket \{Q, D\} \rrbracket\} \\ &= \{s \mid \llbracket T \rrbracket(\llbracket S \rrbracket(s)) \in \llbracket \{Q, D\} \rrbracket\} \\ &= \{s \mid \llbracket S \rrbracket(s) \in \{s' \mid \llbracket T \rrbracket(s') \in \llbracket \{Q, D\} \rrbracket\}\} \\ &= \{s \mid \llbracket S \rrbracket(s) \in wp(T, \{Q, D\})\} \\ &= wp(S, wp(T, \{Q, D\})). \end{aligned}$$

#### 4.3.2. Weakest preconditions of a conditioning construct

We now turn to the conditioning construct. We start with the easy case where the conditioned body does not modify the context expressions.

**Proposition 14.** Assume  $(S, \{Q, D \wedge B\})$  is plain. If

$$wp(S, \{Q, D \wedge B\}) = \{P, C\}$$

then

$$wp(\text{where } B \text{ do } S \text{ end}, \{Q, D\}) = \{P, D\}.$$

The proof uses an additional technical lemma. It expresses that the activity context is left unchanged by a program. It may thus be captured by the same boolean vector expression as soon as its variables are not changed by the program.

**Lemma 15.** Let  $(S, \{Q, D\})$  be plain. Assume

$$\models \{P, C\} S \{Q, D\} \text{ and } \forall s \in \llbracket \{P, C\} \rrbracket : conv(S, s).$$

Then  $\llbracket \{P, C\} \rrbracket = \llbracket \{P, D\} \rrbracket$ . In particular, we have

$$\models \{P, D\} S \{Q, D\} \text{ and } \forall s \in \llbracket \{P, D\} \rrbracket : conv(S, s).$$

**Proof.** Assume  $\sigma \models P$ . Let  $c = \sigma(C)$  and  $c' = \sigma(D)$ . By hypothesis, we have  $conv(S, (\sigma, c))$  and  $\llbracket S \rrbracket(\sigma, c) = (\sigma', c) \in \llbracket \{Q, D\} \rrbracket$ . We deduce  $\sigma'(D) = c$ . Since  $Var(D) \cap Change(S) = \emptyset$ , we have  $c' = \sigma(D) = \sigma'(D) = c$ , too. We deduce  $\{P, C\} \Leftrightarrow \{P, D\}$ , and thus  $\llbracket \{P, C\} \rrbracket = \llbracket \{P, D\} \rrbracket$ .

Without the above assumption of convergence, this lemma is not true. Consider for instance

$$S \equiv \text{loop } X = 0 \text{ do } Y := Y \text{ end}$$

We have

$$\models \{\exists u : X|_u = 0, X = 0\} S \{\forall u : X|_u \neq 0, X \neq 0\}$$

but

$$\not\models \{\exists u : X|_u = 0, X \neq 0\} S \{\forall u : X|_u \neq 0, X \neq 0\}.$$

As an important consequence of the preceding lemma, we obtain the following result.

**Lemma 16** (Extension Lemma). *Assume  $(S, \{Q, D\})$  is plain. If*

$$wp(S, \{Q, D\}) = \{P, C\},$$

*then*

$$wp(S, \{Q, D\}) = \{P, D\}.$$

We can now give the proof of Proposition 14.

**Proof.** Let  $(\sigma, c) \in wp(\text{where } B \text{ do } S \text{ end}, \{Q, D\})$ . By definition,

$$\llbracket \text{where } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = (\sigma', c) \in \llbracket \{Q, D\} \rrbracket,$$

with  $\sigma'$  such that  $(\sigma', c \wedge \sigma(B)) = \llbracket S \rrbracket(\sigma, c \wedge \sigma(B))$ . As  $(\text{Var}(D) \cup \text{Var}(B)) \cap \text{Change}(S) = \emptyset$ , we have

$$\sigma'(D) = \sigma(D) = c \quad \text{and} \quad \sigma'(B) = \sigma(B).$$

Thus,  $c \wedge \sigma(B) = \sigma'(D \wedge B)$ , and  $(\sigma', c \wedge \sigma(B)) \in \llbracket \{Q, D \wedge B\} \rrbracket$ . By assumption, we have  $(\sigma, c \wedge \sigma(B)) \in \llbracket \{P, C\} \rrbracket$ , and  $\sigma \models P$ . Thus,  $(\sigma, c) \in \llbracket \{P, D\} \rrbracket$  as wanted.

Conversely, let  $(\sigma, c) \in \llbracket \{P, D\} \rrbracket$ . Observe first that, by Lemma 16,

$$wp(S, \{Q, D \wedge B\}) = \{P, C\} = \{P, D \wedge B\}.$$

Thus,  $\llbracket S \rrbracket(\sigma, c \wedge \sigma(B)) = (\sigma', c') \in \llbracket \{Q, D \wedge B\} \rrbracket$ , and  $\sigma' \models Q$ . Thus,  $\llbracket \text{where } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = (\sigma', c)$ . As  $\text{Var}(D) \cap \text{Change}(S) = \emptyset$ ,  $\sigma'(D) = \sigma(D) = c$ , and we have  $(\sigma', c) \in \llbracket \{Q, D\} \rrbracket$  as wanted.

To remove the restriction on variables in expression  $B$ , let us consider cases where  $\text{Var}(B) \cap \text{Change}(S) \neq \emptyset$ . We can then introduce a new variable  $\text{Tmp}$  and transform program where  $B \text{ do } S \text{ end}$  into

$$\text{Tmp} := B; \text{ where } \text{Tmp} \text{ do } S \text{ end}$$

Assume  $wp(S, \{Q, D \wedge \text{Tmp}\}) = \{P, C\}$ . Then, using the preceding results, we can see that

$$wp(\text{Tmp} := B; \text{ where } \text{Tmp} \text{ do } S \text{ end}, \{Q, D\}) = \{P[(D?B : \text{Tmp})/\text{Tmp}], D\}.$$

This transformation can in fact be encapsulated in a single rule for  $wp$ .

**Proposition 17.** Assume  $(S, \{Q, D\})$  is plain and let  $Tmp$  be a (new) variable such that  $Tmp \notin (Var(Q) \cup Var(S) \cup Var(D))$ . If

$$wp(S, \{Q, D \wedge Tmp\}) = \{P, C\}$$

then

$$wp(\text{where } B \text{ do } S \text{ end}, \{Q, D\}) = \{P[B/Tmp], D\}.$$

**Proof.** Let  $(\sigma, c) \in wp(\text{where } B \text{ do } S \text{ end}, \{Q, D\})$ . By definition,

$$\llbracket \text{where } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = (\sigma', c) \in \llbracket \{Q, D\} \rrbracket,$$

with  $\sigma'$  such that

$$\llbracket S \rrbracket(\sigma, c \wedge \sigma(B)) = (\sigma', c \wedge \sigma(B)).$$

We have  $\sigma' \models Q$ , and  $\sigma'(D) = c$ . As  $Var(D) \cap Change(S) = \emptyset$ ,  $\sigma(D) = \sigma'(D) = c$ .

Let  $\sigma_1 = \sigma[Tmp \leftarrow \sigma(B)]$  and  $\sigma'_1 = \sigma'[Tmp \leftarrow \sigma(B)]$ .

By definition, we have  $\sigma_1(Tmp) = \sigma(B) = \sigma'_1(Tmp)$ . As  $Tmp \notin Var(D)$ ,  $\sigma_1(D) = \sigma(D) = c$ , and  $\sigma'_1(D) = \sigma'(D) = c$ . As  $Tmp \notin Var(Q)$  and  $\sigma' \models Q$ ,  $\sigma'_1 \models Q$ , too.

As  $Tmp \notin Var(S)$ ,

$$\llbracket S \rrbracket(\sigma_1, c \wedge \sigma(B)) = (\sigma'_1, c \wedge \sigma(B)) = (\sigma'_1, \sigma'_1(D \wedge Tmp))$$

As  $(\sigma'_1, \sigma'_1(D \wedge Tmp)) \in \llbracket \{Q, D \wedge Tmp\} \rrbracket$ ,  $(\sigma_1, c \wedge \sigma(B)) \in \llbracket \{P, C\} \rrbracket$ . Thus,  $\sigma_1 \models P$ . By the Substitution Lemma,  $\sigma \models P[B/Tmp]$ , and  $(\sigma, c) \in \llbracket \{P[B/Tmp], D\} \rrbracket$ .

Conversely, let  $(\sigma, c) \in \llbracket \{P[B/Tmp], D\} \rrbracket$ . Let  $\sigma_1 = \sigma[Tmp \leftarrow \sigma(B)]$ . By the Substitution Lemma,  $\sigma_1 \models P$ . Also, as  $Tmp$  does not appear in  $D$ ,  $\sigma_1(D) = \sigma(D) = c$ . Thus,

$$(\sigma_1, c \wedge \sigma_1(Tmp)) \models \{P, D \wedge Tmp\}.$$

By Lemma 16, we have  $wp(S, \{Q, D \wedge Tmp\}) = \{P, C\} = \{P, D \wedge Tmp\}$ . Thus, there exists some  $\sigma'_1$  such that

$$\llbracket S \rrbracket(\sigma_1, c \wedge \sigma_1(Tmp)) = (\sigma'_1, c \wedge \sigma_1(Tmp)) \models \{Q, D \wedge Tmp\}.$$

In particular,  $\sigma'_1 \models Q$ . As  $Tmp$  does not appear in  $S$ , we have

$$\llbracket S \rrbracket(\sigma, c \wedge \sigma(B)) = (\sigma', c \wedge \sigma(B)),$$

too, with  $\sigma' = \sigma'_1[Tmp \leftarrow \sigma(Tmp)]$ . As  $\sigma'_1 \models Q$ , and  $Tmp$  does not appear in  $Q$ , we have  $\sigma' \models Q$  as well. By the semantics of the conditioning construct, we finally deduce

$$\llbracket \text{where } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = (\sigma', c) \in \llbracket \{Q, D\} \rrbracket$$

as wanted.

**Theorem 18** (Plain WP for linear  $\mathcal{L}$ -programs are definable). *Let  $S$  be a linear  $\mathcal{L}$  program (that is, without any loop construct), and let  $(S, \{Q, D\})$  be plain. Then, there exists a predicate  $P$  such that  $wp(S, \{Q, D\}) = \{P, D\}$ .*

**Proof.** By induction on the structure of  $S$ . The case of assignment and sequential composition are trivial. Let  $S \equiv \text{where } B \text{ do } T \text{ end}$ . Let  $Tmp$  be a new variable not in  $Var(Q) \cup Var(D) \cup Var(S)$ . By induction hypothesis, there exists an assertion  $\{P, C\}$  such that  $wp(T, \{Q, D \wedge Tmp\}) = \{P, C\}$ . By Proposition 17, we have  $wp(S, \{Q, D\}) = \{P[B/Tmp], D\}$ .

The example at the end of Section 4.1 shows that the theorem is no longer true if the restriction  $Var(D) \cap Change(S) = \emptyset$  is removed. In this case, the set  $wp(S, \{Q, D\})$  cannot be defined by any assertion  $\{P, C\}$  in general.

Yet, we can obtain a *weaker* result as follows. Let  $\#$  be a new variable. We can observe that

$$(\sigma, c) \in \llbracket \{Q, D\} \rrbracket \quad \text{iff} \quad (\sigma[\# \leftarrow c], c) \in \llbracket \{Q \wedge \# = D, D\} \rrbracket.$$

Note then that

$$\llbracket \{Q \wedge \# = D, D\} \rrbracket = \llbracket \{Q \wedge \# = D, \#\} \rrbracket.$$

As  $\#$  is a *new* variable, we are able to apply the previous theorem to  $wp(S, \{Q \wedge \# = D, \#\})$ . It yields some assertion  $\{P, C\}$  which defines this set (observe  $\#$  may occur in  $P$ ). By the Extension Lemma, it is described by  $\{P, \#\}$  as well.

**Theorem 19.** *Let  $\{Q, D\}$  be an assertion, and  $S$  be a linear  $\mathcal{L}$  program. Let  $\#$  be a new variable not in  $Var(D) \cup Var(S) \cup Var(Q)$ . Then, there exists a predicate  $P$  such that*

$$wp(S, \{Q, D\}) = \{(\sigma, c) \mid \sigma[\# \leftarrow c] \models P\}.$$

**Proof.** Let  $\{P, \#\} = wp(S, \{Q \wedge \# = D, \#\})$ . Consider a state  $(\sigma, c)$  such that  $\sigma[\# \leftarrow c] \models P$ . Let  $\sigma_1 = \sigma[\# \leftarrow c]$ . Then  $\sigma_1 \models P$ . Also,  $\sigma_1(\#) = c$ . Thus  $(\sigma_1, c) \in \llbracket \{P, \#\} \rrbracket$ . Let  $(\sigma'_1, c) = \llbracket S \rrbracket(\sigma_1, c)$ . We have  $(\sigma'_1, c) \in \llbracket \{Q \wedge \# = D, \#\} \rrbracket$ . It is then routine to show that  $\llbracket S \rrbracket(\sigma, c) \in \llbracket \{Q, D\} \rrbracket$ . The converse proof is of the same vein.

It is interesting to notice that Theorem 18 is a special case of Theorem 19. Actually, assume that  $Var(D) \cap Change(S) = \emptyset$ . Consider  $wp(S, \{Q, D\})$ , and apply Theorem 19. We have

$$wp(S, \{Q, D\}) = \{(\sigma, c) \mid \sigma[\# \leftarrow c] \models P\}$$

where  $\#$  is a new variable. As  $Var(D) \cap Change(S) = \emptyset$ ,  $\sigma(D) = c$ . This set of states is thus equal to

$$\{(\sigma, c) \mid \sigma[\# \leftarrow \sigma(D)] \models P \wedge \sigma(D) = c\}$$

that is

$$\{(\sigma, c) \mid \sigma \models P[D/\#] \wedge \sigma(D) = c\}$$

that is precisely  $\llbracket \{P[D/\#], D\} \rrbracket$ , as announced in Theorem 18.

#### 4.4. Definability of weakest liberal preconditions

Except for the **where** construct, the basic definability properties of weakest strict preconditions for assignment and sequencing still hold for the weakest liberal preconditions with similar proofs. We concentrate here on the case of the conditioning construct, and we show that no result analogous to Theorem 18 may be expected. Of course, this only concerns non-linear programs. With the two-component assertional proof system, the data-parallel case turns out to be much more complex than the usual sequential one.

The difficulty to be addressed is the following. In presence of divergence, we cannot infer the initial activity contexts from post-assertions. We could expect a property of the form:

If

$$\{P, C \wedge B\} = wlp(S, \{Q, D \wedge B\}),$$

then

$$\{P, C\} = wlp(\text{where } B \text{ do } S \text{ end}, \{Q, D\}).$$

An easy partial result is given by the proof rule for the **where** construct.

**Proposition 20.** *Assume  $\text{change}(S) \cap \text{var}(C) = \emptyset$ . If*

$$\{P, C \wedge B\} \subseteq wlp(S, \{Q, D \wedge B\}),$$

*then*

$$\{P, C\} \subseteq wlp(\text{where } B \text{ do } S \text{ end}, \{Q, C\}).$$

Unfortunately, the preceding proposition does not hold if we replace inclusions by equalities, as shown by the following example.

Consider the following program  $S$  over a one-dimensional domain  $\mathcal{D} = [1..M]$ . Intuitively, the value of  $X|_u$  is initially set to false at each active index. Then, the values of  $X|_1, X|_2$ , etc. are repeatedly fetched, until the value false is found. Remember that we assume that function  $\text{stoi}$  is defined everywhere, so that fetching makes sense for any address (think for instance of some cyclic numbering scheme as in MPL).

```

X := False;
where This = 1 do
  N := 1; X := True;

```

```

loop  $X|_N$  do
   $N := N + 1$ 
end
end

```

Define

$$P \equiv (\forall u \neq 1 : X|_u = \text{true}) \quad \text{and} \quad C \equiv (\text{This} = 1).$$

**Fact 21.**  $S$  diverges from  $(\sigma, c)$  iff  $(\sigma, c) \in \llbracket \{P, C\} \rrbracket$ .

**Proof.**  $S$  diverges iff index 1 is active on entering the loop, and no  $X|_u$  is spotted to be false. Because of the initial assignment  $X := \text{False}$ ,  $X|_u$  is false iff index  $u$  is initially active. Thus, all indices  $u \neq 1$  have to be idle initially for divergence to occur. That is, the initial context is described by  $\text{This} = 1$ . Also, each  $X|_u$  has to be true initially (except for  $u = 1$ ).

The first crucial observation is now that the value of  $C$  does not depend on the environment.

**Fact 22.**  $wlp(S, \{\text{true}, C\}) = \{\text{true}, C\}$ .

**Proof.** Assume  $(\sigma, c) \in \llbracket \{\text{true}, C\} \rrbracket$ . If  $S$  diverges from this state, then we are done. If  $S$  converges, then the final context is the same as the initial one, and we are done again.

Conversely, let  $s$  such that  $\llbracket S \rrbracket(s) \in \llbracket \{\text{true}, C\} \rrbracket$ . If  $S$  diverges from this state, then  $s \in \llbracket \{P, C\} \rrbracket \subseteq \llbracket \{\text{true}, C\} \rrbracket$ . If it converges, then its context is still described by  $C$ , as  $C$  does not depend on the environment.

The second crucial observation is that the context described by  $C$  is not identically active.

**Fact 23.**  $wlp(\text{where } C \text{ do } S \text{ end}, \{\text{true}, \text{True}\})$  is not definable by any assertion.

**Proof.** Fix an environment  $\sigma \models P$ . Then both  $(\sigma, \text{True})$  and  $(\sigma, \sigma(C))$  belong to  $wlp(\text{where } C \text{ do } S \text{ end}, \{\text{true}, \text{True}\})$ . As  $\sigma(C) \neq \text{True}$ , this set cannot be described by any assertion, as remarked in Section 4.1.

Yet, the definability property holds for the weakest liberal preconditions of the *where* construct, *modulo* the set of divergent states.

**Proposition 24.** Assume  $\text{Var}(D) \cap \text{Change}(S) = \emptyset$  and let  $\text{Tmp}$  be a (new) variable such that  $\text{Tmp} \notin (\text{Var}(Q) \cup \text{Var}(S) \cup \text{Var}(D))$ . If

$$wlp(S, \{Q, D \wedge \text{Tmp}\}) \subseteq \llbracket \{P, D \wedge \text{Tmp}\} \rrbracket \subseteq wlp(S, \{Q, D \wedge \text{Tmp}\})$$

then

$$\begin{aligned} & wp(\text{where } B \text{ do } S \text{ end}, \{Q, D\}) \\ & \subseteq \llbracket P[B/Tmp], D \rrbracket \\ & \subseteq wlp(\text{where } B \text{ do } S \text{ end}, \{Q, D\}). \end{aligned}$$

**Proof.** The first inclusion is a part of Proposition 14.

The proof of the second inclusion is very similar to the corresponding proof for the weakest preconditions.

Consider a state  $(\sigma, c) \models \{P[B/Tmp], D\}$ . Let  $\sigma_1 = \sigma[Tmp \leftarrow \sigma(B)]$ . By the Substitution Lemma,  $\sigma_1 \models P$ . If  $S$  diverges from the state  $(\sigma, c \wedge \sigma(B))$ , then the result trivially holds. Otherwise, let  $c_1 = c \wedge \sigma(B)$ . There exists an environment  $\sigma'$  such that  $\llbracket S \rrbracket(\sigma, c_1) = (\sigma', c_1)$ . As  $Tmp$  does not appear in  $S$ , we have  $\llbracket S \rrbracket(\sigma_1, c_1) = (\sigma'_1, c_1)$  too, with  $\sigma'_1 = \sigma'[Tmp \leftarrow \sigma(Tmp)]$ . By definition, we have  $\sigma(B) = \sigma_1(Tmp)$ . As  $Tmp$  does not appear in  $D$ ,  $\sigma_1(D) = \sigma(D) = c$ . Thus,

$$(\sigma_1, c_1) \models \{P, D \wedge Tmp\}.$$

We deduce  $\llbracket S \rrbracket(\sigma_1, c_1) \models \{Q, D \wedge Tmp\}$ . In particular,  $\sigma'_1 \models Q$ . As  $Tmp$  does not appear in  $Q$ ,  $\sigma' \models Q$  as well. As  $Var(D) \cap Change(S) = \emptyset$ ,  $\sigma'(D) = \sigma(D) = c$ . By the semantics of the conditioning construct, we finally have

$$\llbracket \text{where } B \text{ do } S \text{ end} \rrbracket(\sigma, c) = (\sigma', c) \models \{Q, D\}.$$

#### 4.5. Discussion: Extending the assertion language?

So far, we have shown that the plain weakest preconditions of linear programs are always definable by some two-component assertion, but that the weakest liberal preconditions of non-linear program are not in general. The two-component assertion language is not expressive enough to denote its own weakest preconditions.

In Theorem 19, we have shown an alternative to the description of weakest preconditions. By introducing some auxiliary variable  $Aux$ , which denotes the context, we can find a predicate  $P$  which denotes the weakest preconditions. This auxiliary variable  $Aux$  is precisely the counterpart of the symbol  $\#$  introduced by Le Guyadec and Viot in [14]. In [22], it is shown this theorem is true in all cases, including weakest liberal preconditions of non-linear programs (this uses a Gödel encoding of computations, very much as in the completeness proof of the classical Hoare's logic in [1]).

Going back to the counterexamples of Section 4.1, reconsider the first program

$$S_1 \equiv \text{loop } True \text{ do } X := X \text{ end}$$

we have

$$wlp(S_1, \{true, True\}) = \{(\sigma, c) \mid \sigma[Aux \leftarrow c] \models \exists u: Aux|_u = true\}.$$



For the second program, let

$$S_2 \equiv X := X + 1, \quad Q \equiv (\forall u: X|_u = 1) \vee (\forall u: X|_u = 2), \quad D \equiv (X = 2).$$

We have

$$\begin{aligned} wlp(S_2, \{Q, D\}) = \{(\sigma, c) \mid \sigma[Aux \leftarrow c] \models \\ Q \wedge [(\forall u: Aux|_u = true) \vee (\forall u: Aux|_u = false)]\}. \end{aligned}$$

Moreover, we have a logical link between valid specifications and weakest preconditions denoted by a predicate with some auxiliary variable. It is stated in the following property (see [22] for further details), which generalizes the consequence of Lemma 8.

**Proposition 25.** *Let  $S$  be a  $\mathcal{L}$ -program,  $\{Q, D\}$  be an assertion. Let  $W$  be a predicate and  $Aux$  be a new variable not in  $Var(S) \cup Var(Q) \cup Var(D)$ , such that*

$$wlp(S, \{Q, D\}) \{(\sigma, c) \mid \sigma[Aux \leftarrow c] \models W\}.$$

*For any assertion  $\{P, C\}$  such that*

$$\models \{P, C\} S \{Q, D\}$$

*we have*

$$\models (P \wedge (\forall u: C|_u = Aux|_u)) \Rightarrow W$$

This direction has been explored by Cachera and Utard in [7].

## 5. Completeness of the proof system

We now want to establish the completeness for our proof system. We restrict ourself to linear  $\mathcal{L}$ -programs. It is well known that proving completeness in presence of a loop requires some complex machinery: invariant predicates, variant expressions, etc. (see [1] for instance), which would obscure the main line of our work at this point.

The proof of the completeness is constructed in an incremental way. We start from a basic case: completeness for a restricted form of specification (restrictions on the program and on the assertions.) We introduce *auxiliary variables* and a rule to handle them, which allows us to remove all restrictions step by step. The following notion is the restricted form of programs we first consider.

**Definition 26 (Regular program).** A program  $P$  is *regular* if, for any subprogram of  $P$  of the form *where  $B$  do  $S$  end*, we have  $Var(B) \cap Change(S) = \emptyset$ .

The results of the previous section can be restated as follows. As we assume the specifications to be plain and the programs to be regular, we call it *restricted* definability.

**Proposition 27** (Restricted definability of WP of regular programs). *Let  $S$  be a regular, linear  $\mathcal{L}$  program, and let  $(S, \{Q, D\})$  be plain. Then, there exists an assertion  $\{P, C\}$  such that*

$$\llbracket \{P, C\} \rrbracket = wp(S, \{Q, D\}).$$

*In particular,  $\models \{P, C\} S \{Q, D\}$ .*

We aim at proving the following theorem.

**Theorem 28** (Restricted completeness, plain specifications, regular, linear programs). *Let  $\{P, C\} S \{Q, D\}$  be a plain specification, where  $S$  is a regular, linear program. If*

$$\models \{P, C\} S \{Q, D\}$$

*then*

$$\vdash \{P, C\} S \{Q, D\}.$$

**Proof.** The proof of this theorem follows the lines of [1]. It uses the weakest preconditions calculus. For any regular, linear program  $S$  and any plain pair  $(S, \{Q, D\})$ , there exists some assertion  $\{P', C'\}$  such that  $\llbracket \{P', C'\} \rrbracket = wp(S, \{Q, D\})$ . Using the Consequence Rule, it suffices to demonstrate that  $\vdash \{wp(S, \{Q, D\})\} S \{Q, D\}$ .

The proof is done by induction on the structure of the regular, linear program  $S$ , using the Definability Properties of Section 4.3.

The cases of the assignment and sequencing constructs are straightforward. Let us consider in more details the case of the conditioning construct, with  $S \equiv \text{where } B \text{ do } T \text{ end}$ . As  $S$  is regular by hypothesis, we have  $\text{Change}(T) \cap \text{Var}(B) = \emptyset$ . As the specification is plain, we have  $\text{Change}(S) \cap \text{Var}(D) = \emptyset$ . As  $\text{Change}(T) = \text{Change}(S)$ , we also have  $\text{Change}(T) \cap \text{Var}(D) = \emptyset$ . The Definability Property yields an assertion  $\{P, C\}$  such that  $\{P, C\} = wp(T, \{Q, D \wedge B\})$ . By the Extension Lemma, we get  $wp(T, \{Q, D \wedge B\}) = \{P, D \wedge B\}$ .

Program  $T$  is regular and linear as  $S$  is so. Specification  $\{P, D \wedge B\} T \{Q, D \wedge B\}$  is plain. Thus, the induction hypothesis yields

$$\vdash \{P, D \wedge B\} T \{Q, D \wedge B\}.$$

As  $(\text{Var}(B) \cup \text{Var}(D)) \cap \text{Change}(T) = \emptyset$ , the *where* Rule of the proof system applies, and we get

$$\vdash \{P, D\} \text{ where } B \text{ do } T \text{ end } \{Q, D\}.$$

Furthermore, the Definability property gives

$$wp(\text{where } B \text{ do } T \text{ end}, \{Q, D\}) = \{P, D\}.$$

Hence the desired result:

$$\vdash \{wp(\text{where } B \text{ do } T \text{ end}, \{Q, D\})\} \text{ where } B \text{ do } T \text{ end } \{Q, D\}.$$

Consider now a plain specification  $\models \{P, C\} S \{Q, D\}$ , with  $S$  being a regular, linear program. By the Definability Property, there exists some assertion  $\{P', C'\}$  such that  $\{P', C'\} = wp(S, \{Q, D\})$ . By the above result, we know that  $\vdash \{P', C'\} S \{Q, D\}$ . By the Consequence Lemma 8, we get that  $\{P, C\} \Rightarrow \{P', C'\}$ . We can thus apply the Consequence Rule of the proof system. It yields  $\vdash \{P, C\} S \{Q, D\}$  as wanted.

This demonstrates the completeness of the proof system for plain specifications and regular, linear programs.

### 5.1. Extending the proof of completeness to non-regular, linear programs

In the presence of non-regular programs, we are no longer able to find any assertion that expresses the weakest preconditions. Thus, we first have to transform a non-regular program into a regular one. This can be done by introducing an *auxiliary variable*, which stores the value of the vector boolean expression: program

where  $B$  do  $S$  end

is transformed into

$Tmp := B; \text{ where } Tmp \text{ do } S \text{ end}$

Using such a variable, can be interpreted as keeping track of the nested activity context in a stack. Each new variable  $Tmp$  is a frame of the stack.

But, instead of transforming programs in order to be able to prove them, we claim that it is possible to *encapsulate* this transformation into the proof system itself. The notion corresponding to the syntactic *auxiliary* variable is that of a semantic *hidden* variable in assertions.

**Rule 6** (Elimination of hidden variables). *Let  $E$  be any vector expression:*

$$\frac{\{P, C\} S \{Q, D\}, \quad Tmp \notin Var(S) \cup Var(Q) \cup Var(D)}{\{P[E/Tmp], C[E/Tmp]\} S \{Q, D\}}.$$

We denote by  $\vdash^* \{P, C\} S \{Q, D\}$  that a specification formula is derivable in the  $\vdash$  proof system augmented with this new rule.

The soundness of the extended proof system  $\vdash^*$  is expressed by the following proposition.

**Theorem 29** (Soundness of  $\vdash^*$ ). *The  $\vdash^*$  proof system is sound: if*

$$\vdash^* \{P\} S \{Q\}$$

then

$$\models \{P\} S \{Q\}.$$

**Proof.** As  $\vdash^*$  is an extension of  $\vdash$  with the Elimination Rule, it suffices to check the following fact. Let us consider  $Tmp \notin \text{Var}(S) \cup \text{Var}(Q) \cup \text{Var}(D)$  and  $E$  an expression. If  $\models \{P, C\} S \{Q, D\}$ , then  $\models \{P[E/Tmp], C[E/Tmp]\} S \{Q, D\}$ .

Assume  $\models \{P, C\} S \{Q, D\}$ .

Let us consider  $(\sigma, c) \models \{P[E/Tmp], C[E/Tmp]\}$ . In particular,  $\sigma \models P[E/Tmp]$ . Let  $\sigma_1 = \sigma[Tmp \leftarrow \sigma(E)]$ . By the Substitution Lemma,  $\sigma_1 \models P$ . Moreover,  $\sigma_1(C) = \sigma(C[E/Tmp]) = c$ . Thus,  $(\sigma_1, c) \models \{P, C\}$ .

By hypothesis, we have  $\llbracket S \rrbracket(\sigma_1, c) = (\sigma'_1, c)$ , with  $(\sigma'_1, c) \models \{Q, D\}$ .

Finally, let  $(\sigma', c) = \llbracket S \rrbracket(\sigma, c)$ . As  $Tmp \notin \text{Var}(S)$ , we have  $\sigma'_1 = \sigma'[Tmp \leftarrow \sigma(E)]$ . What is more,  $\sigma'_1 \models Q$  and  $Tmp \notin \text{Var}(Q)$ , so  $\sigma' \models Q$ , and  $\sigma'_1(D) = c$  with  $Tmp \notin \text{Var}(D)$ , so that  $\sigma'(D) = c$ . Thus, we have  $(\sigma', c) \models \{Q, D\}$ . Thus,  $\models \{P[E/Tmp], C[E/Tmp]\} S \{Q, D\}$ .

We now want to establish the following completeness theorem.

**Theorem 30** (Restricted completeness, plain specifications, linear program). *Let  $\{P, C\} S \{Q, D\}$  be a plain specification, with  $S$  a linear program. If*

$$\models \{P, C\} S \{Q, D\}$$

then

$$\vdash^* \{P, C\} S \{Q, D\}.$$

Note that Proposition 17 already used new “hidden” variables to guarantee the definability of the weakest preconditions of any plain specification, as expressed in Theorem 18. We can now prove Completeness Theorem 30 for non-regular programs.

**Proof.** The proof is similar to the one of the Completeness Theorem 28 for regular programs. It uses a structural induction on  $S$ . The only new case to consider is  $S \equiv \text{where } B \text{ do } T \text{ end}$ , with  $\text{Var}(B) \cap \text{Change}(S) \neq \emptyset$ . Pick up a “new” variable  $Tmp$  such that  $Tmp \notin \text{Var}(S) \cup \text{Var}(Q) \cup \text{Var}(D)$ . Such a variable exists because the expressions from the program and from the assertion language are finite terms. By Theorem 18, we know there exists some assertion

$$\{P, C\} = wp(T, \{Q, D \wedge Tmp\}).$$

We have  $\text{Var}(D \wedge Tmp) \cap \text{Change}(S) = \emptyset$  by the choice of  $Tmp$ . Thus,  $wp(T, \{Q, D \wedge Tmp\}) = \{P, D \wedge Tmp\}$  by the Extension Lemma. By the induction hypothesis, we have

$$\vdash^* \{P, D \wedge Tmp\} T \{Q, D \wedge Tmp\}.$$

We also have  $\{P \wedge B = \text{Tmp}, D \wedge B\} \Rightarrow \{P, D \wedge \text{Tmp}\}$ . We can thus apply the Consequence Rule. This yields

$$\vdash^* \{P \wedge B = \text{Tmp}, D \wedge B\} T \{Q, D \wedge \text{Tmp}\}.$$

Then, we apply the where Rule, and we get

$$\vdash^* \{P \wedge B = \text{Tmp}, D\} \text{ where } B \text{ do } T \text{ end } \{Q, D\}.$$

Thanks to the Consequence Rule, this rewrites into

$$\vdash^* \{P[B/\text{Tmp}] \wedge B = \text{Tmp}, D\} \text{ where } B \text{ do } T \text{ end } \{Q, D\}.$$

Finally, applying the Elimination Rule with  $E \equiv B$  yields

$$\vdash^* \{P[B/\text{Tmp}], D\} \text{ where } B \text{ do } T \text{ end } \{Q, D\}.$$

According to Proposition 17,  $wp(S, \{Q, D\}) = \{P[B/\text{Tmp}], D\}$ . Thus

$$\vdash^* wp(S, \{Q, D\}) S \{Q, D\}.$$

As before, we conclude the proof with Lemma 8, the Consequence Rule and the Definability Property.

### 5.2. Extending the proof of completeness to non-plain specifications

We now focus on general specifications, where  $\text{Var}(D) \cap \text{Change}(S)$  may be not empty. Surprisingly enough, the Elimination Rule is sufficient to prove the completeness in this case, and there is no need of any other additional rule.

**Theorem 31** (Completeness, linear programs). *Let  $S$  be a linear program. If*

$$\models \{P, C\} S \{Q, D\}$$

*then*

$$\vdash^* \{P, C\} S \{Q, D\}.$$

**Proof.** Assume  $\models \{P, C\} S \{Q, D\}$ . As the expressions of the assertion language are finite terms, there exists a “new” hidden variable  $\text{Tmp}$  such that  $\text{Tmp} \notin \text{Var}(S) \cup \text{Var}(Q) \cup \text{Var}(D)$ . Let us show that

$$\models \{P \wedge \text{Tmp} = C, C\} S \{Q \wedge \text{Tmp} = D, \text{Tmp}\}.$$

Let  $(\sigma, c)$  be in  $\llbracket \{P \wedge \text{Tmp} = C, C\} \rrbracket$ . We have in particular  $(\sigma, c) \models \{P, C\}$ . By hypothesis, we thus get  $\llbracket S \rrbracket(\sigma, c) = (\sigma', c) \models \{Q, D\}$ .

Furthermore, we have  $\sigma(\text{Tmp}) = \sigma(C) = c$ . As  $\text{Tmp} \notin \text{Var}(S)$ , we have  $\sigma'(\text{Tmp}) = \sigma(\text{Tmp}) = c$ , and  $(\sigma', c) \models \{Q, D\}$  gives  $\sigma'(D) = c$ . We conclude that  $(\sigma', c) \models \{Q \wedge \text{Tmp} = D, \text{Tmp}\}$ .

As  $Tmp \notin \text{Var}(S)$ , we are in the case of a plain specification, so the Completeness Theorem 30 applies and yields

$$\vdash^* \{P \wedge Tmp = C, C\} S \{Q \wedge Tmp = D, Tmp\}.$$

As  $\{Q \wedge Tmp = D, Tmp\} \Rightarrow \{Q, D\}$ , we can apply the Consequence Rule. It yields

$$\vdash^* \{P \wedge Tmp = C, C\} S \{Q, D\}.$$

Applying then the Elimination Rule with  $E \equiv C$  yields

$$\vdash^* \{P \wedge C = C, C\} S \{Q, D\}.$$

Finally, as  $\{P, C\} \Rightarrow \{P \wedge C = C, C\}$ , we deduce by another application of the Consequence Rule that

$$\vdash^* \{P, C\} S \{Q, D\}.$$

## 6. A two-phase proof methodology for $\mathcal{L}$ programs

A crucial step remains to be made for a practical application of our results. Quoting Apt and Olderog's seminal book [1, Section 3.4]:

*“Formal proofs are tedious to follow. We are not accustomed to following a line of reasoning presented in small, formal steps [...].*

*A possible strategy lies in the facts that [programs] are structured. The proof rules follow the syntax of the program, so the structure of the program can be used to structure the correctness proof. We can simply present the proof by giving a program with assertions interleaved at appropriate places [...].*

*This type of proof is more simple to study and analyze than the one we used so far. Introduced by Gries and Owicki, it is called a Proof Outline.”*

The presentation of Apt and Olderog focuses on control-parallel programs, i.e., sequential processes composed with the  $\parallel$  operator. We show here that the approach of Gries and Owicki can be adapted as well to data-parallel  $\mathcal{L}$  programs, giving birth to a notion of data-parallel annotations. We present a simple proof method that allows, after a first step that slightly transforms the program, to handle it as an usual scalar program.

The first step consists of a labeling of the program that expresses the depth of conditioning constructs. In other words, a subprogram labeled by  $i$  is executed within the scope of  $i$  where constructs. This labeling follows the syntax of the program: labels are increased on entering the body of a new conditioning construct. Context expressions are saved here in a series of auxiliary variables. This allows us to alleviate any restriction on context expressions of conditioning constructs.

The second step consists of a proof method similar to that used in the scalar case, interleaving assertions and program constructs.

### 6.1. First step: syntactic labeling

In this step, we associate to each subprogram of the considered program an integer label that counts the number of nesting where constructs. Counting starts at 0 for the whole program. Consider for instance the program

```

where  $X > 0$  do
   $X := X + 1$ ;
  where  $X > 2$  do
     $X := X + 1$ ;
  end
end

```

We want to get the following labeling:

```

(0) where  $X > 0$  do
  (1)  $X := X + 1$ ;
  (1) where  $X > 2$  do
    (2)  $X := X + 1$ 
  end
end

```

In order to store context expressions, we distinguish particular auxiliary variables that do not appear in programs.

**Definition 32.** Variables  $\{Tmp_i \mid i \in \mathbb{N}\}$  are such that for any program  $S$ , and for any index  $i$ ,  $Tmp_i \notin Var(S)$ . They are called *auxiliary variables*.

The conditioning construct can be seen as a stack mechanism: entering a *where* construct is the same as pushing a value on a context stack, while exiting this construct corresponds to a “pop”. The label is namely the height of the stack. At a given point, the current context is corresponding to the conjunction of all the stack’s values. Each auxiliary variable is used to store one frame of the context stack. Thanks to this storage, the variables appearing in context expressions may be modified. We can thus alleviate restrictions on context expressions of conditioning constructs.

For a subprogram at depth  $i$ , the current context is the current value of  $Tmp_0 \wedge \dots \wedge Tmp_i$ . To get a clearer presentation of this fact, we add annotations of the form  $[Tmp_i \equiv B]$  to each *where* construct. The previous example is recast into

```

(0) where  $X > 0$  do  $[Tmp_1 \equiv X > 0]$ 
  (1)  $X := X + 1$ ;
  (1) where  $X > 2$  do  $[Tmp_2 \equiv X > 2]$ 
    (2)  $X := X + 1$ 
  end
end

```

$$\begin{array}{c}
\frac{\forall j > i, Tmp_j \notin Var(Q)}{\{Q[\bigwedge_{k=0}^i Tmp_k ? E: X/X]\} (i) \ X := E \ \{Q\}} \\
\\
\frac{\{P\} S^* \{R\} \quad \{R\} T^* \{Q\} \quad \forall j > Lab(S), Tmp_j \notin Var(R) \cup Var(Q)}{\{P\} S^* ; \{R\} T^* \{Q\}} \\
\\
\frac{P \Rightarrow P' \quad \{P'\} S^* \{Q'\} \quad Q' \Rightarrow Q \quad \forall j > Lab(S), Tmp_j \notin Var(Q) \cup Var(Q')}{\{P\}\{P'\} S^* \{Q'\}\{Q\}} \\
\\
\frac{\{P\} S^* \{Q\} \quad Lab(S) = i + 1 \quad \forall j > i, Tmp_j \notin Var(Q)}{\{P[B/Tmp_{i+1}]\} (i) \ \text{where } B \text{ do } [Tmp_{i+1} \equiv B] \{P\} S^* \{Q\} \text{ end } \{Q\}} \\
\\
\frac{\{P\} S^* \{Q\}}{\{P\} S^{**} \{Q\}}
\end{array}$$

where  $S^{**}$  is obtained from  $S^*$  by deleting any assertion.

Fig. 4. Rules for annotation.

Labeling is thus made by induction on the program's syntactic structure, running over the program's syntactic tree in a depth-first manner. For the entire program, counting starts at 0 for the labels and at 1 for the auxiliary variables,  $Tmp_0$  denoting the initial context the program is executed in. If  $T$  is a labeled subprogram of a program  $S$ , we denote by  $Lab(T)$  the outer label associated to  $T$ .

## 6.2. Second step: proof outline

As we use labeled programs, and auxiliary variables to store contexts, we know the expression denoting the current context at each place in the program. We can then drop context expressions out of assertions and proceed exactly the same way as in the scalar case, with backward substitutions. The only differences are that expressions in substitutions are conditioned by a conjunction of  $Tmp_k$  and that the data-parallel **where** construct adds a new substitution. The rules for inserting assertions in proof outlines are given in Fig. 4. Contiguity between two assertions refers to the use of the Consequence Rule. If  $S$  is a labeled subprogram, we denote by  $S^*$  a proof outline obtained from  $S$  by insertion of assertions.

Let us explain intuitively the need of restrictions of the form " $\forall j > i, Tmp_j \notin Var(Q)$ ". In the rule for the conditioning construct, we substitute  $Tmp_{i+1}$  with  $B$ . We thus need that  $Tmp_{i+1} \notin Var(Q)$  to respect the conditions of the Substitution Rule. But, as the postcondition ( $Q$ ) is the same for  $S$  and for **where**  $B$  **do**  $S$  **end**,



```

      {((Tmp0 ∧ X > 0 ∧ (Tmp0 ∧ X > 0?X + 1:X) > 2?
        (Tmp0 ∧ X > 0?X + 1:X) + 1:
        (Tmp0 ∧ X > 0?X + 1:X))|u = 4}

(0) where X>0 do [Tmp1 ≡ X > 0]
      {((Tmp0 ∧ Tmp1 ∧ (Tmp0 ∧ Tmp1?X + 1:X) > 2?
        (Tmp0 ∧ Tmp1?X + 1:X) + 1:
        (Tmp0 ∧ Tmp1?X + 1:X))|u = 4}

(1) X:=X+1;
      {((Tmp0 ∧ Tmp1 ∧ X > 2?X + 1:X)|u = 4}

(1) where X>2 do [Tmp2 ≡ X > 2]
      {((Tmp0 ∧ Tmp1 ∧ Tmp2?X + 1:X)|u = 4}

(2) X:=X+1
      {X|u = 4}

end
      {X|u = 4}

end
      {X|u = 4}

```

Fig. 5. The program annotated with its proof outline.

we need this condition to be satisfied for every nesting depth greater than  $Lab(S)$ .

### 6.3. A simple example

We go back to our previous example. We want to prove the two following specifications.

<pre> {X <sub>u</sub> = 2, True} where X &gt; 0 do   X := X+1;   where X &gt; 2 do     X := X+1   end end {X <sub>u</sub> = 4, True} </pre>	<pre> {X <sub>u</sub> = 1, True} where X &gt; 0 do   X := X+1;   where X &gt; 2 do     X := X+1   end end {X <sub>u</sub> = 2, True} </pre>
---	---

These specifications mean that, if the initial value of  $X$  at index  $u$  is 2, then its final value after execution of the program will be 4 at the same index, and if it is 1, then the final value will be 2. The proofs are simply done by establishing the following proof outline – the result of the first step has already been given as example in the previous section.

**First proof.** If we denote by  $P$  the first assertion of this proof outline displayed on Fig. 5, we only have to prove that

$$X|_u = 2 \wedge Tmp_0 = True \Rightarrow P.$$

In other words, we prove that

$$X|_u = 2 \Rightarrow P[True/Tmp_0].$$

The predicate  $P[True/Tmp_0]$  is equivalent to

$$(X > 0 \wedge (X > 0?X + 1 : X) > 2?$$

$$(X > 0?X + 1 : X) + 1;$$

$$(X > 0?X + 1 : X))|_u = 4.$$

Let us consider an index  $u$  such that  $X|_u = 2$ . Then, the boolean expression  $(X > 0)|_u$  is true. As  $X + 1|_u > 2$ ,  $((X > 0?X + 1 : X) > 2)|_u$  is also true.

Conditional expression

$$(X > 0 \wedge (X > 0?X + 1 : X) > 2?$$

$$(X > 0?X + 1 : X) + 1;$$

$$(X > 0?X + 1 : X))|_u.$$

thus simplifies into  $(X > 0?X + 1 : X) + 1|_u$ , which in turn simplifies into  $X + 1 + 1|_u$ .

Assertion  $P[True/Tmp_0]$  thus simplifies into  $X + 1 + 1|_u = 4$ , which is true.

**Second proof.** As no simplification using the value of  $X$  occurs in the first proof outline, the second is almost the same: we just replace the value 4 by the value 2. Then, if we denote by  $P'$  the assertion obtained by substituting 4 by 2 in  $P$ , we just have to check that

$$X|_u = 1 \Rightarrow P'[True/Tmp_0].$$

Let us consider an index  $u$  such that  $X|_u = 1$ . Then, the boolean expression  $(X > 0)|_u$  is true. But this time, as  $X + 1|_u = 2$ ,  $((X > 0?X + 1 : X) > 2)|_u$  is false.

Conditional expression

$$(X > 0 \wedge (X > 0?X + 1 : X) > 2?$$

$$(X > 0?X + 1 : X) + 1;$$

$$(X > 0?X + 1 : X))|_u$$

thus simplifies into  $(X > 0?X + 1 : X)|_u$ , which in turn simplifies into  $X + 1|_u$ .

Assertion  $P'[True/Tmp_0]$  thus simplifies into  $X + 1|_u = 2$ , which is true.

## 7. Conclusion and related work

We have presented an assertional approach towards the formal validation of data-parallel programs. This has been done by defining a small language called  $\mathcal{L}$ , which is intended to be a common kernel of real data-parallel languages, and giving it a formal semantics. Thanks to the macroscopic viewpoint for data parallelism (see the general presentation in [2]), this is a streamline extension of the method used for Pascal-like scalar languages [1]. The key of the method is the definition of state assertions. We propose to use two-component assertions, in which the first component specifies the current value of parallel variables, and the second one describes the current extent of parallelism through a parallel boolean expression. We have presented an extension of Hoare's logic for these two-component assertions. The proof system is quite similar to those used in the case of scalar languages: it follows the same structure and respects semantic properties such as compositionality.

Yet, handling two-component assertions brings an additional level of complexity. First, the assertion language suffers from a restricted expressive power, as it is not closed under classical propositional operations. The Definability Property does not hold for weakest liberal preconditions. Second, auxiliary variables are necessary to store the successive values of the extent of parallelism.

In fact, this illustrates a well-known drawback of assertional proof systems: this kind of semantics is not well-suited to denote control flow properties of parallel programs. For instance, Owicki and Gries introduce auxiliary variables to catch the control flow information in the proof of parallel programs [16]. This cannot be avoided to prove properties like mutual exclusion. Introducing auxiliary variables makes the study of completeness more difficult than in the scalar case. As in the work of Owicki and Gries, additional rules have to be considered to introduce and then erase these variables.

It is interesting to discuss whether the additional complexity of the formalism is inherent to data-parallel languages, or is rather due to our technical choices in handling the extent of parallelism. Actually, the alternative is to consider one-component assertions, in which a distinguished variable describes the extent of parallelism within vector predicates. The assertions are then usual predicates and the constraints on the extent of parallelism are directly handled by assertions.

This direction was already pointed out by Narayana and Clint in their study of the formal validation of Actus programs [8]. They called this distinguished variable  $\#$ . This technique leads to a much simpler sequential-like proof system, as shown by Virot and Le Guyadec [14]. The idea is simply to transform conditional **where** blocks into sequences of assignments to the  $\#$  variable. Using this techniques, any (linear)  $\mathcal{L}$  program can be transformed into a sequence of conditioned or unconditioned (as introduced in [6]) assignments. The resulting proof system is well-suited for (semi)-automatic verification of programs following the method of *verification conditions* proposed by Gordon [11].

In fact, the discussion of Section 4.5 shows that any auxiliary variable can locally play the role of the distinguished variable  $\#$ . This leads to a new kind of assertion,

in which an additional variable *Aux* is used to denote context. We have shown that  $wlp(S, \{Q, D\})$  can always be defined by such an assertion. Moreover, it is shown in [7] that the whole approach can be reworked out with this new notion of assertion, yielding a simpler proof of completeness.

In spite of its higher technical difficulty, the two-component assertion approach appears as a useful framework in deriving explicit proofs of data-parallel programs. Two-component assertions provide an explicit description of the extent of parallelism, which allows the programmer to factorize the proof into three different phases: first, label all statements with their extent of parallelism; second, derive the predicates satisfied by the parallel variables at each point; third, check the global consistency of these predicates with respect to the extent of parallelism.

It is possible to extend this work to other data-parallel languages. An extension of the  $\mathcal{L}$  language is described in [4]. It defines a *data-parallel escape construct*. This new construct extends the data-parallel break and continue constructs found in real languages like HyperC [17] or MPL [15]. The natural semantics handles the activity by a *multi-context* mechanism. In [5], the two-component assertion language is extended to handle multi-context, which leads to a similar proof system for  $\mathcal{L}$ -programs with escape constructs. An interesting direction would be to extend this work to data-parallel languages that take into account notions of data alignment and mapping.

## Acknowledgements

We wish to thank Joaquim Gabarró, Alan Stewart, Bjørn Lisper and Guy-René Perrin for their useful comments on this work.

## References

- [1] K.R. Apt, E.-R. Olderog, Verification of Sequential and Concurrent Programs, Text and Monographs in Computer Science, Springer, Berlin, 1991.
- [2] L. Bougé, The Data-Parallel Programming Model: A Semantic Perspective, in: G.-R. Perrin, A. Darte (ed.), the Data-Parallel Programming Model, Lecture Notes in Computer Science, vol. 1132, Tutorial Series, Springer, Berlin, 1996, pp. 4–26.
- [3] L. Bougé, Y. Le Guyadec, G. Utard, B. Virot, A proof system for a simple data-parallel programming language, in: C. Girault, (ed.), Proc. IFIP WG 10.3, Applications in Parallel and Distributed Computing, Caracas, Venezuela, North-Holland, Amsterdam, 1994, pp. 63–72.
- [4] L. Bougé, J.-L. Levaire, Control structures for data-parallel SIMD languages: semantics and implementation, Future Generation Computer Systems 8 (1992) 363–378.
- [5] L. Bougé, G. Utard, Escape constructs in data-parallel languages: semantics and proof system, Research Report 94-18, LIP, ENS Lyon, 1994; URL: <http://www.ens-lyon.fr/LiP>.
- [6] D. Cachera, Two completeness results about a proof system for simple data-parallel language, Research Report 95-29, LIP, ENS Lyon, 1995; URL: <http://www.ens-lyon.fr/LiP>.
- [7] D. Cachera, G. Utard, Proving data-parallel programs: a unifying approach, Parallel Process. Lett. 1996, 6 (4) (1996) 491–505.
- [8] M. Clint, K.T. Narayana, On the completeness of a proof system for a synchronous parallel programming language, in: 3rd Conf. Found. Softw. Techn. and Theor. Comp. Science, Bangalore, India, December 1983.

- [9] DPCE Subcommittee Numerical C Extensions Group of X3J11, Data-Parallel C Extension, ANSI, December 1994.
- [10] J. Gabarró, R. Gavalda, An approach to correctness of data-parallel algorithms, *J. Parallel and Distrib. Comput.* 22 (2) (1994) 185–201.
- [11] M.J.C. Gordon, *Programming Language Theory and its Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [12] W.D. Hillis, G.L. Steele Jr., Data parallel algorithms, *Comm. ACM* 29 (12) (1986) 1170–1183.
- [13] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* 12 (10) (1969) 576–583.
- [14] Y. Le Guyadec, B. Viro, Sequential-like proofs of data-parallel programs, *Parallel Process. Lett.* (1996) 6 (3) (1996) 415–426.
- [15] MasPar Computer Corporation, Sunnyvale, CA, *Maspar Parallel Application Language Reference Manual*, 1990.
- [16] S. Owicki, D. Gries, Verifying properties of parallel programs: an axiomatic approach, *Comm. ACM* 19 (5) (1976) 279–285.
- [17] N. Paris, *HyperC specification document*, Technical Report 93-1, HyperParallel Technologies, École Polytechnique, Palaiseau, France, 1993.
- [18] R.H. Perrot, A language for array and vector processors, *ACM Trans. Progr. Lang.* 1 (2) (1979) 177–195.
- [19] A. Stewart, Reasoning about data-parallel array assignment, *J. Parallel Distrib. Comput.* 27 (1985) 79–85.
- [20] A. Stewart, An axiomatic treatment of SIMD assignment, *BIT* 30 (1990) 70–82.
- [21] Thinking Machine Corporation, Cambridge, MA, *C\* programming guide*, 1990.
- [22] G. Utard, *Semantics of data-parallel languages. Applications to validation and compilation*, Ph.D. thesis, École Normale Supérieure de Lyon, 1995 (in French); URL: <http://www.ens-lyon.fr/LiP>.